Chapter - two

Process Management

Process and Threads

Lecture 2.1

Outline

- Process concepts
- The process model
- Process Management
- Types of Processes
- Process creation and termination
- Process states and PCB
- Threads
- Process scheduling algorithms
- Inter process communication

Process concepts

- Process vs. Program
  - Program
    - It is sequence of instructions defined to perform some task
    - It is a passive entity
  - Process
    - It is a program in execution
    - It is an instance of a program running on a computer
    - It is an active entity
    - A processor performs the actions defined by a process

The process model

- A process is just an executing program.
- The CPU switches back and forth from process to process,
- this rapid switching back and forth is called multiprogramming.
- However, at any instant of time, the CPU runs only one program.
- Thus giving the users illusion of parallelism.

System calls

- The interface between the operating system and the user program is defined by the set system calls that the operating system provides.

- System calls provide the interface between a running program and the operating system.
  - Generally available as assembly-language instructions.
  - Languages defined to replace assembly language for systems
  - Programming allow system calls to be made directly (e.g., C, C++)

System calls...

- Three general methods are used to pass parameters between a running program and the operating system.
  - Pass parameters in registers.
  - Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
  - Push (store) the parameters onto the stack by the program, and pop off the stack by operating system.
- Types of system calls
  - Process control
  - File management
  - Device management
  - Information maintenance
  - Communications

Process Management

- Overview
- The most fundamental task of modern operating systems is process management.
- It includes:
  - Creation of processes
  - Allocation of resources to processes
  - Protecting resources of each processes from other processes
  - Enabling processes to share and exchange information
  - Enabling synchronization among processes for proper sequencing and coordination when dependencies exist
  - Termination of processes

Types of Processes

- There are two types of processes:
  - Sequential Processes
    - Execution progresses in a sequential fashion, i.e. one after the other
    - At any point in time, at most one process is being executed
  - Concurrent Processes
    - There are two types of concurrent processes
    - True Concurrency (Multiprocessing)
      - » Two or more processes are executed simultaneously in a multiprocessor environment
      - » Supports real parallelism
    - Apparent Concurrency (Multiprogramming)
      - » Two or more processes are executed in parallel in a uniprocessor environment by switching from one process to another
      - » Supports pseudo parallelism, i.e. fast switching among processes gives illusion of parallelism

Process creation

- Operating system needs some way to make sure all the necessary processes are created and terminated.

- There are four principal events that cause a processes to be created:

  1. System initialization
  2. Execution of a process creation system call by a running process.
  3. A user request to create a new process.
  4. Initialization of a batch of job.

Process creation...

1. System initialization - When an operating system is booted, typically several processes are created.

- Some of these are foreground processes, (i.e. processes that interact with users and perform work for them.)
- Others are background processes, which are not associated with particular users.
  - Example: one background process may be designed to accept incoming-mail sleeping most of the time, incoming request for web pages
- Processes that stay in the background to handle some activities are called daemons.
- **In linux: ps, ps –fl, ps -efl**
- **In windows: ctrl-alt-del**

Process creation...

2- Creation of processes by running process

- Often a running process will issue system calls to create one or more new processes to help it to do its job.

Example: If a large amount of data is being fetched over a network for subsequent processing,

- one process to fetch the data and put them in a shared buffer,
- while the second process removes the data item and process them.

Process creation...

3 - A user request to create processes

- Users can start a program by typing a command or (double) clicking an icon.
- Taking either of these actions starts a new process and runs the selected program in it.

Process creation...

4 - Initiation of a batch of job

- Here user can submit batch of jobs to the system (possibly remotely) in main frame computer.
- When the operating system decides that it has the resources to run another job, it creates a new process and runs the next job from the input queue in it.

- In all these cases, a new process is created by having an existing process execute a process creation system call.

Process creation: Summary

- One process can create another process, perhaps to do some work for it
- The original process is called the parent
- The new process is called the child
- The child is an (almost) identical copy of parent (same code, same data, etc.)
- The parent can either:
  - wait for the child to complete, or
  - continue executing in parallel (concurrently) with the child
- There are also two possibilities in terms of the address space of the new process:
  - the child process is a duplicate of the parent process
  - the child process has a new program loaded to it

Process creation: Summary...

- In Linux, a process creates a child process using the system call fork( )
- In child process, fork( ) returns 0
- In parent process, fork( ) returns process id of new child
- Child often uses exec( ) to start another completely different program

- In windows, a single system call, CreateProcess, handles both
  - process creation and
  - loading the correct program in to the new process.

Process creation in Linux

//Example: process0.c

```
#include <sys/types.h>
#include <stdio.h>
int main(){
int id;
 printf("demo on process creation");
 id=fork();
  if(id > 0) {
 /*parent process*/
  wait(NULL);
printf("This is parent process [process id:%d].\n",getpid());
}
else if (id==0) {
/* child process*/
printf("fork child process[process id:%d].\n",getpid());
printf("fork parent process id:%d].\n",getppid());
}
Else {
 /*for creation failed!*/
Printf("fork creation failed\n");
  }
  return 0;
 }
```

Process creation in Linux...

//Example: process.c

```
#include <sys/types.h>
#include <stdio.h>
int a = 10;
 int main(void)
{
int b;
pid_t pid;/*process id */
b = 100;
printf("before fork\n");
    pid = fork();
if(pid == 0){ /* child */
      a++; b++;
}
else /* parent */
wait(pid);
printf("after fork,
a = %d, b = %d\n",
a, b);
   exit(0);
   }
```

Process creation...

//Example: process2.c

```
#include <stdio.h>
#include <sys/types.h>

int main() {
  int id,i;
 printf("Start of main \n");
     id=fork();
       if(id>0) {
      wait(NULL);
  /*parent process*/
printf("Parent is running\n");
      }
    else if(id==0)
          {
   /*child process*/
printf("\n fork created...\n");
 }
  else {
/*fork creation failed*/
printf("\n fork creation
failed!\n");
 }
printf("Printing the
numbers from 1 to 5\n");
for(i=1; i<=5; i++)
printf("%d ",i);
printf("\n");
printf("End of the main
function...\n");
return 0;
   }
```

Process creation...

//Example: process3.c

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
int main( ){
pid_t child_pid;
// Create a new child process;
child_pid = fork();
if (child_pid < 0) {
printf("forkfailed");
return 1;
}
else if (child_pid == 0) {
printf ("child process
successfullycreated!\n");
printf ("child_PID = %d,parent_PID
= %d\n",getpid(), getppid());
}
```

```
  else {
wait(NULL);
printf ("parent process
success.created!\n");
printf ("child_PID = %d,
parent_PID = %d", getpid()
, getppid());
}
return 0;
}
```

Process Termination

- After a process has been created, it starts running and does whatever its job is.

- Sooner or later the new process will terminate, due to one of the following conditions:

  1. Normal exit (voluntary)
  2. Error exit (voluntary)
  3. Fatal error (involuntary)
  4. Killed by another process (involuntary)

Process Termination...

1- Normal exit – Most process terminates because they have done their work.

Example:

- When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished.
- This call is Exit in UNIX and ExitProcess in windows.
- Screen oriented programs also support voluntary termination.
  - Word processors, Internet browser and similar programs always have an icon or menu item that the user can click to tell the process to terminate. File → Exit

Process Termination...

2- Error exit - When the process discovers an error.

- Example: If the user typed a command

```
javac Function.java
```

- To compile the program and no such file exists, the compiler simply exits.

3- Fata error – is an error caused by the process, often due to a program bug.

- Example:
  - Executing an illegal instruction
  - Referencing non-existent memory
  - Dividing by zero

Process Termination...

4 - Killed by another process – a process executes a system call telling the operating system to kill some other process.

- In UNIX this call is Kill.
- In window TerminateProcess.

Process Hierarchies

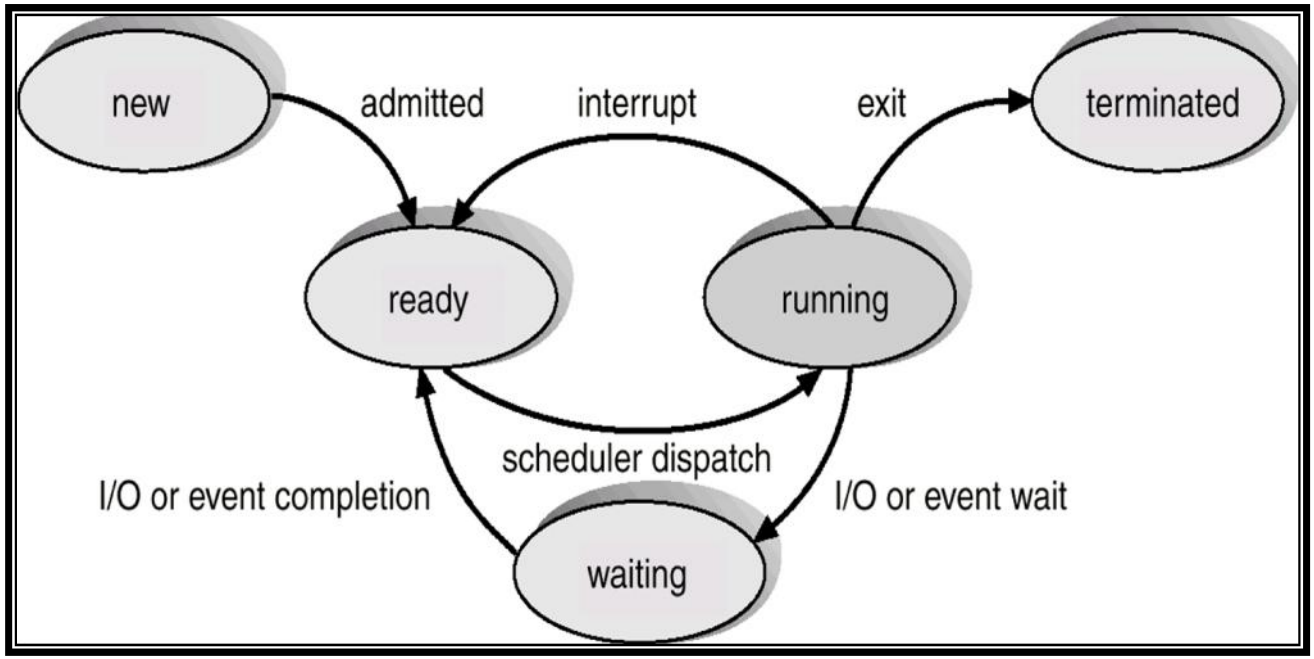
- For various reasons, a process can creates another process.
- The child process can itself create more processes, forming a process hierarchy.
- Example: How Linux initializes itself when it is started.
  - A special process, called `init` is present in the boot image.
  - It reads a file telling how many terminals there are
  - Then it forks off one new process per terminal
  - These processes waiting for something to log in
  - If log in is successful, the login process executes a shell to accept commands.
  - These commands may start up more processes.....
- All processes created above belong to a single tree with `init` at the root. pstree
- Windows doesn't have any process hierarchy.

Process States

- During its lifetime, a process passes through a number of states.

- new: The process is being created.
- ready: The process is waiting to be assigned to a processor.
- running: Instructions are being executed.
- Waiting/blocked: The process is waiting for some event to occur such as I/O operation
- terminated: The process has finished execution.

Diagram of Process State



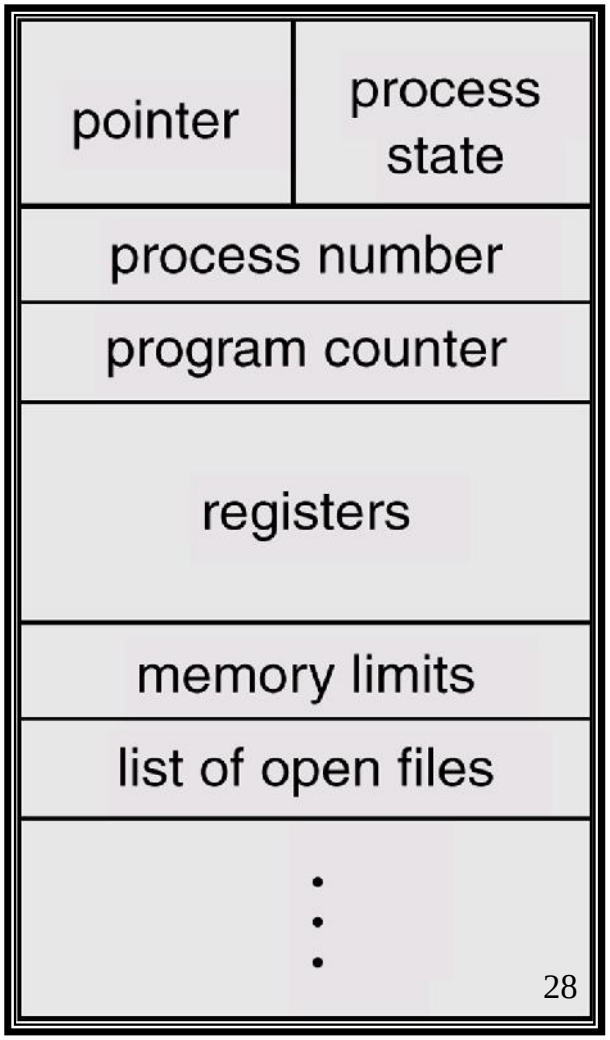
State Transitions in Five-State Process Model

- new → ready
  - Admitted to ready queue; can now be considered by CPU scheduler
- ready → running
  - CPU scheduler chooses that process to execute next, according to some scheduling algorithm
- running → ready
  - Process has used up its current time slice
- running → blocked
  - Process is waiting for some event to occur (for I/O operation to complete, etc.)
- blocked → ready
  - Whatever event the process was waiting on has occurred

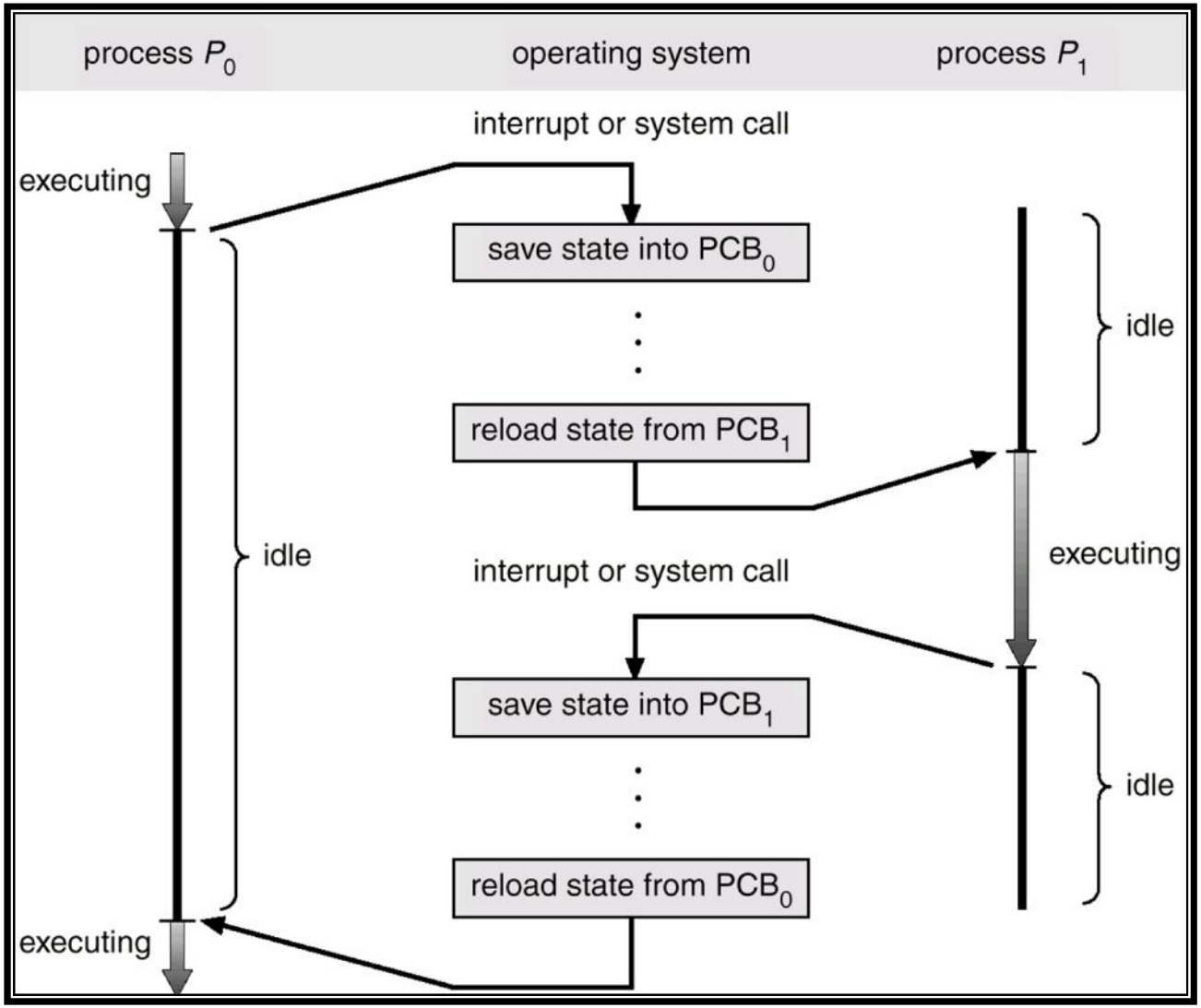
Process Control Block (PCB)

Information associated with each process.

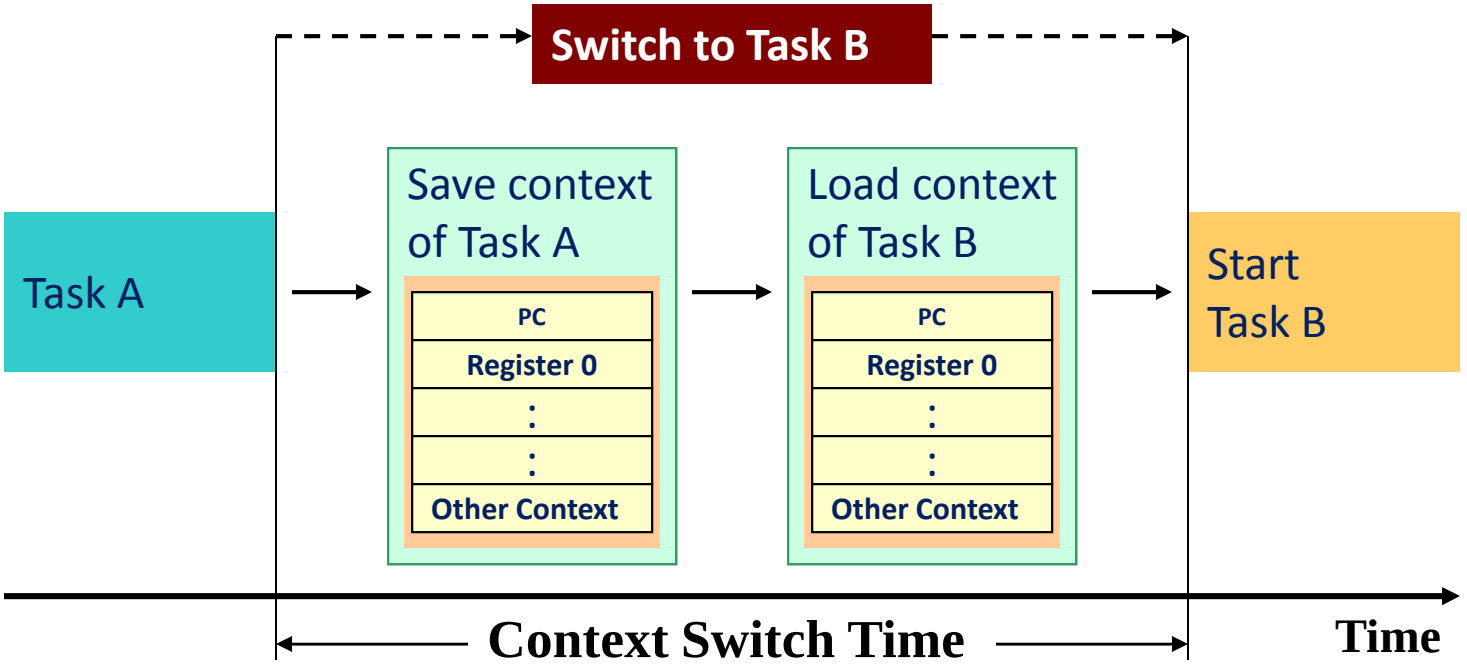
- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



CPU Switch From Process to Process



CPU Switch From Process to Process
Context Switch Time



Threads

- Thread vs. Process
  - A thread is a dispatchable unit of work (lightweight process) that has independent context, state and stack
  - A process is a collection of one or more threads and associated system resources
  - Traditional operating systems are single-threaded systems
  - Modern operating systems are multithreaded systems

Threads...

- The thread has:
  - A program counter that keeps track of which instruction to execute next.
  - Has registers which hold its current working variables
  - Has a stack which contains the execution history

- Processes are used to group resources together.
- Threads are the entities scheduled for execution on the CPU.

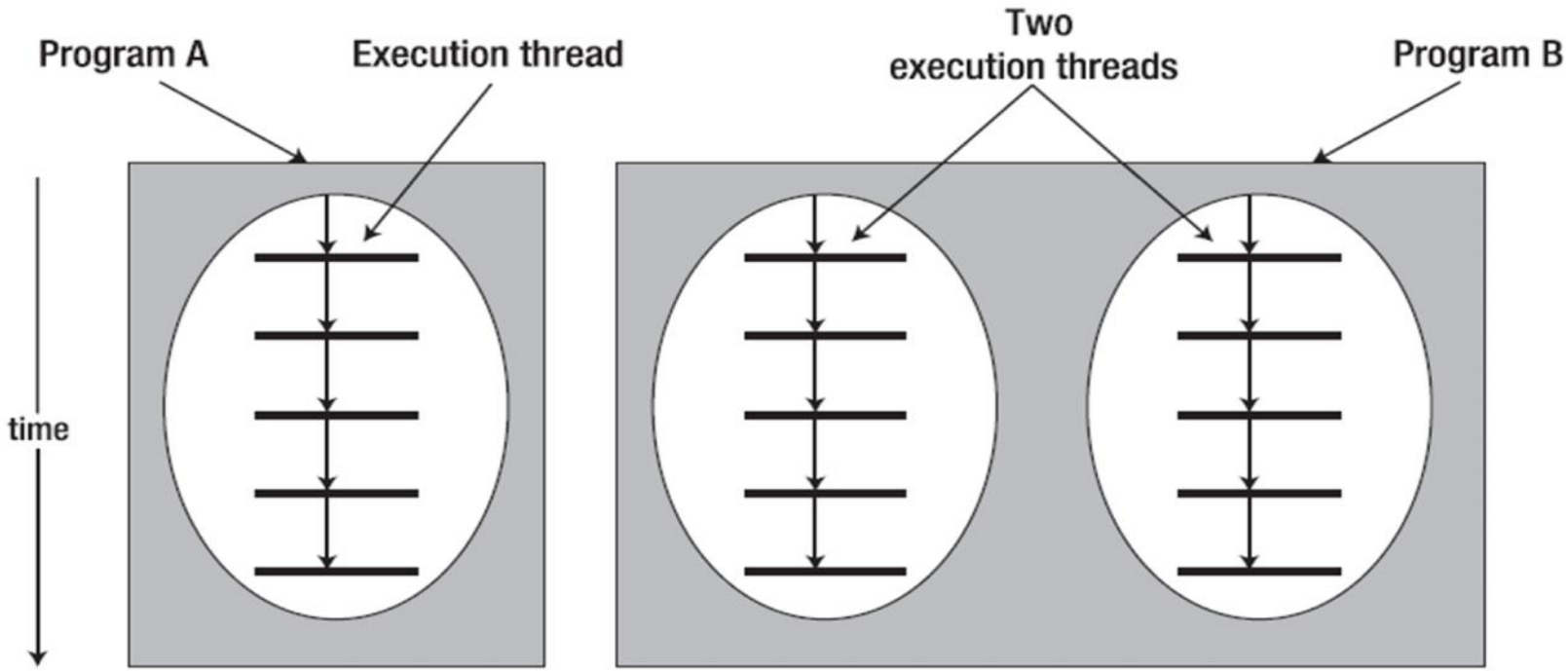
Threads...

- What threads add to the process model is to allow multiple executions to take place in the same process environment.

- Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer.
  - In the former case the threads share an address space, open files, and other resources.
  - In the later case processes share physical memory, disks, printers and other resources.

Threads...

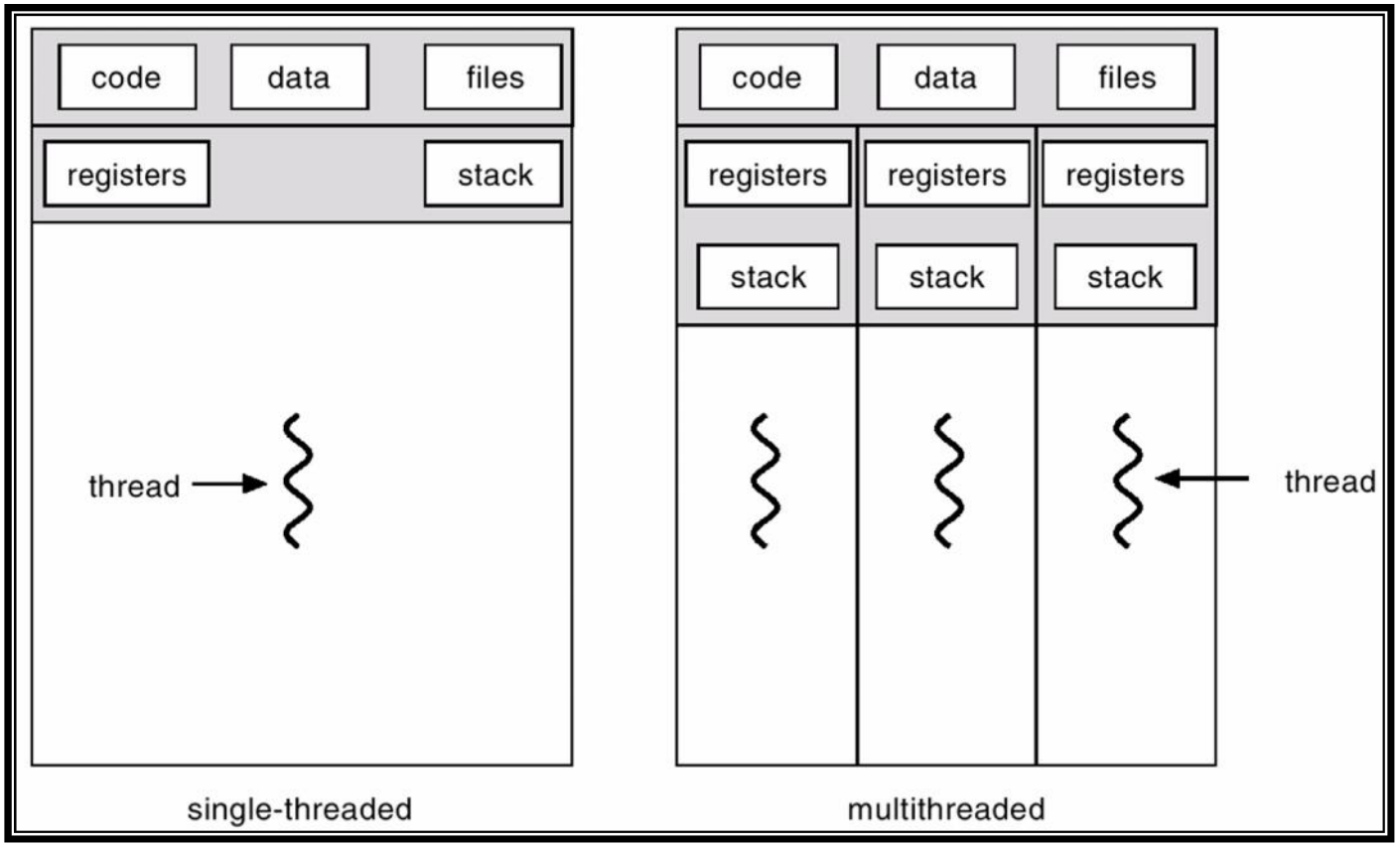
- Threads are sometimes called lightweight process.
- Multithreading- is to describe the situation of allowing multiple threads in the same process.



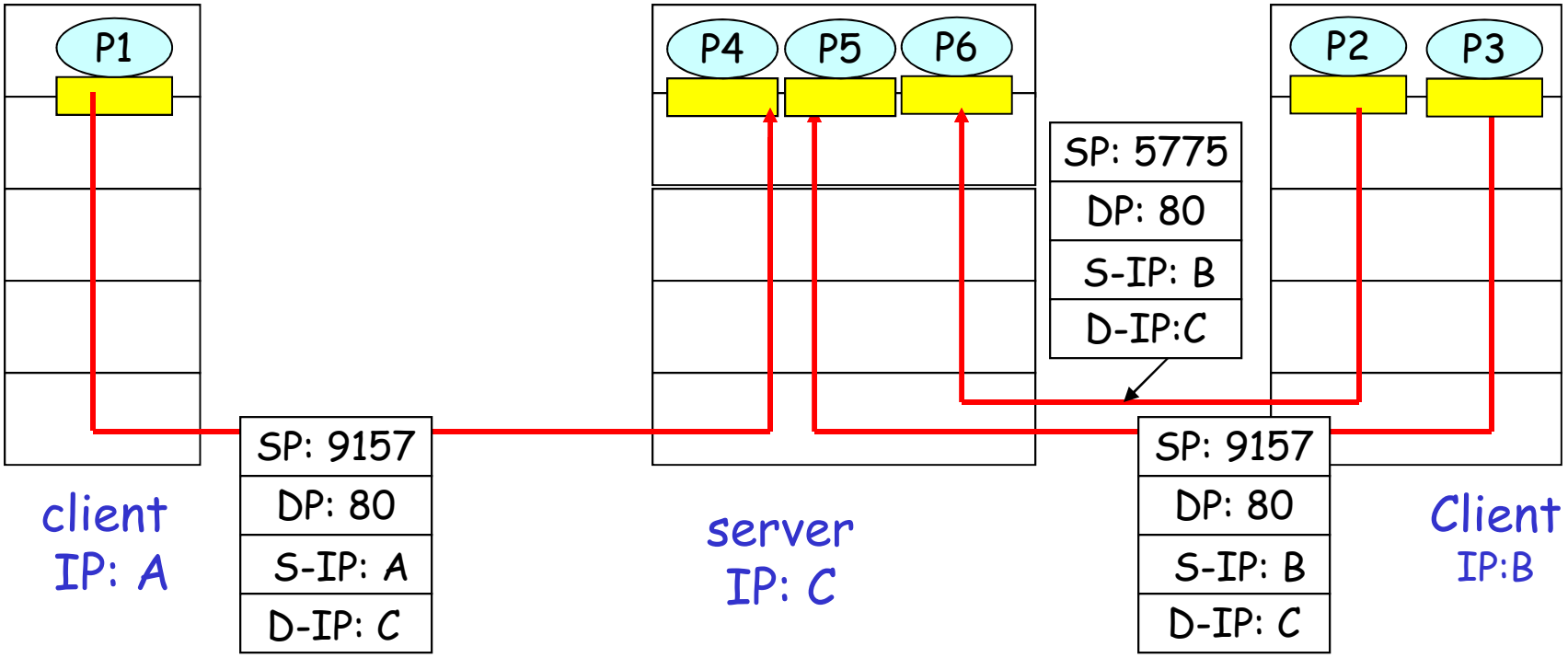
Single and Multithreaded Processes...

- A thread is a single sequence of execution within a program
- Multithreading refers to multiple threads of control within a single program
  - each program can run multiple threads of control within it, e.g., Web Browser, MS Words,...

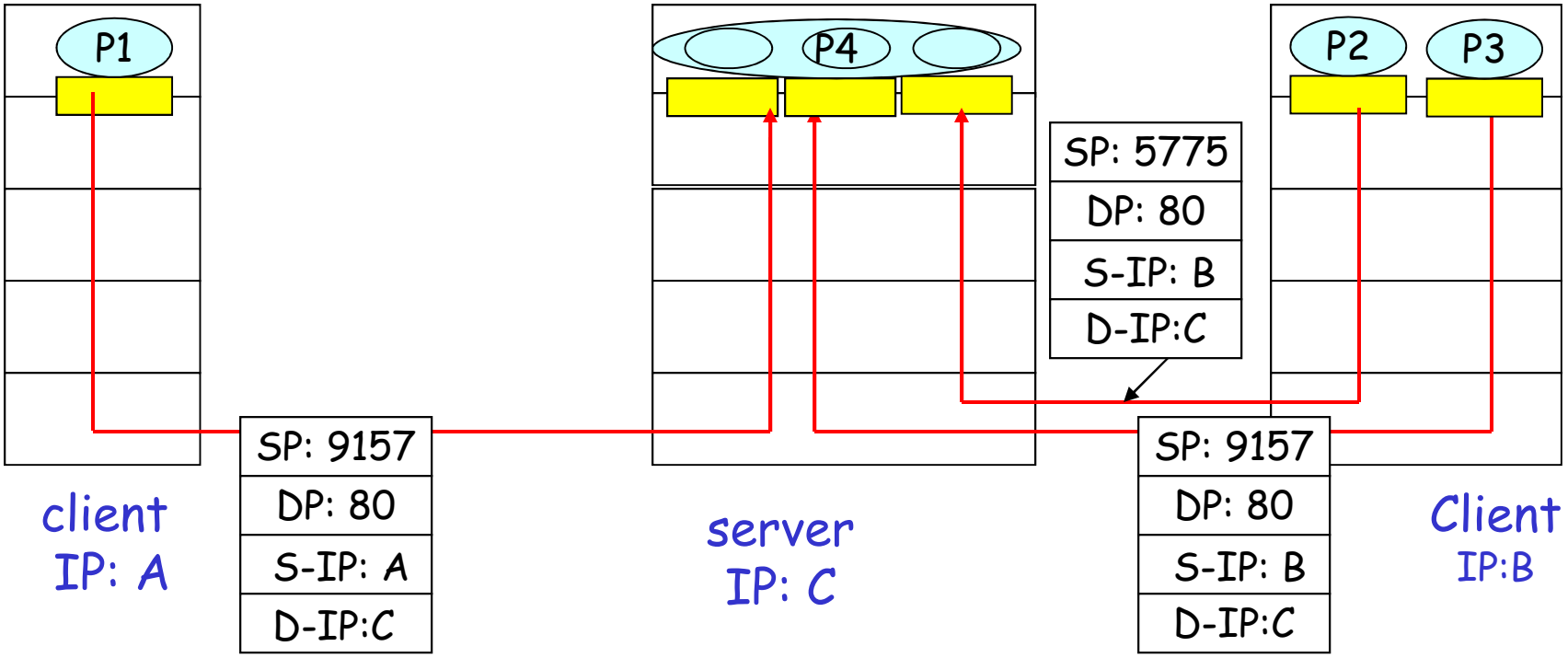
Single and Multithreaded Processes...



Single threaded Web Server



Multi-Threaded Web Server



Thread Types

User Threads

- Thread management done by user-level threads library

- Examples
  - POSIX Pthreads
  - Mach C-threads
  - Solaris threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows 95/98/NT/2000/XP...
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

Multithreading Models

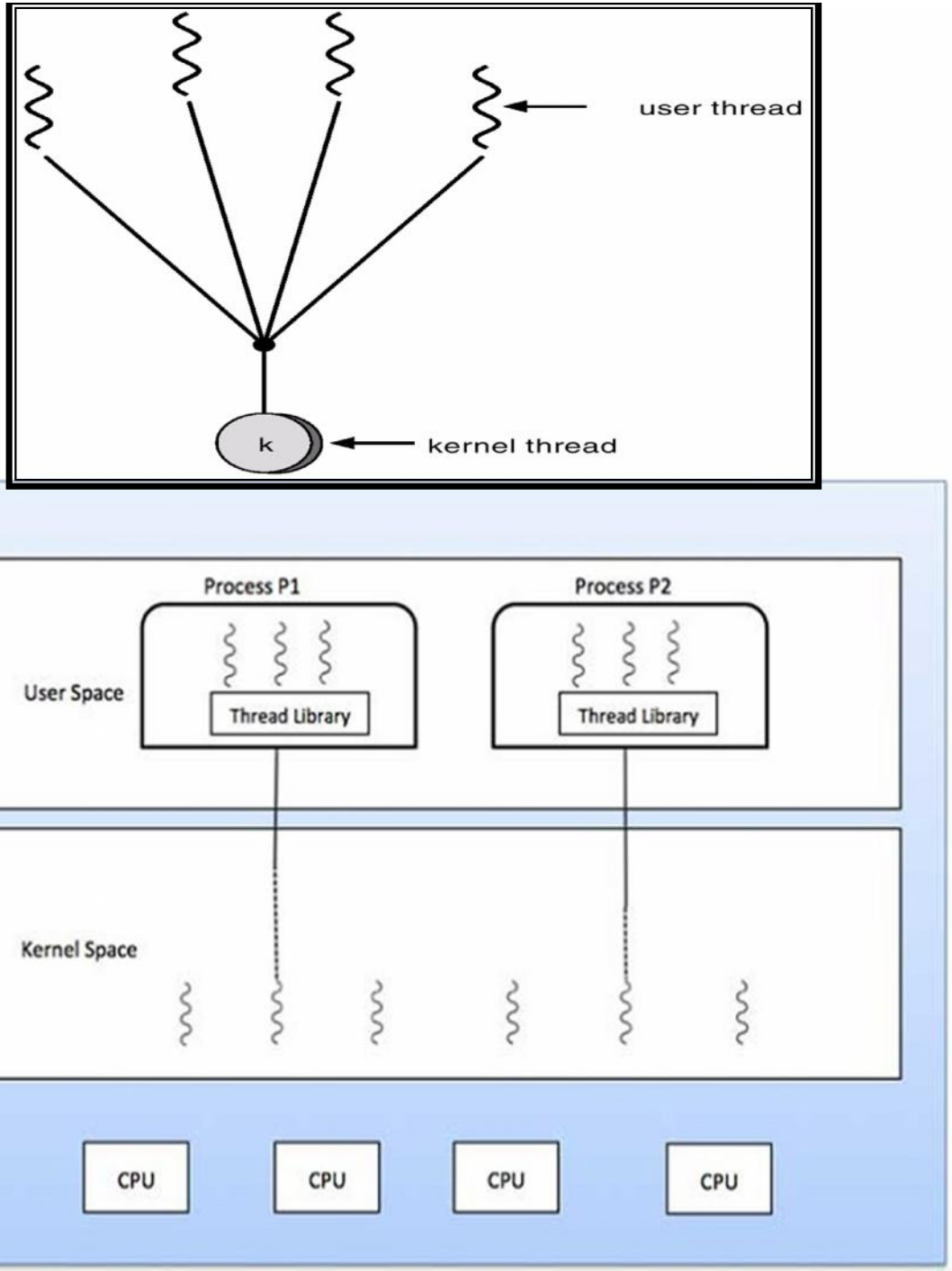
- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread.
- Thread management is done in user space by the thread library.

Drawbacks:

- When thread makes a blocking system call, the entire process will be blocked.
- Only one thread can access the Kernel at a time,
  - so multiple threads are unable to run in parallel on multiprocessors.



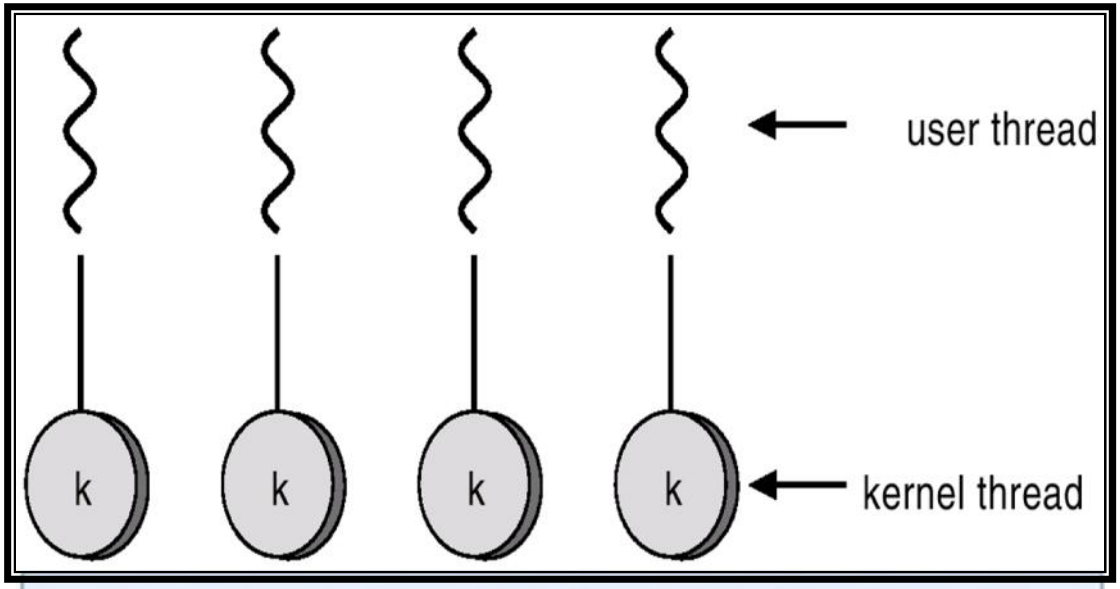
One-to-One

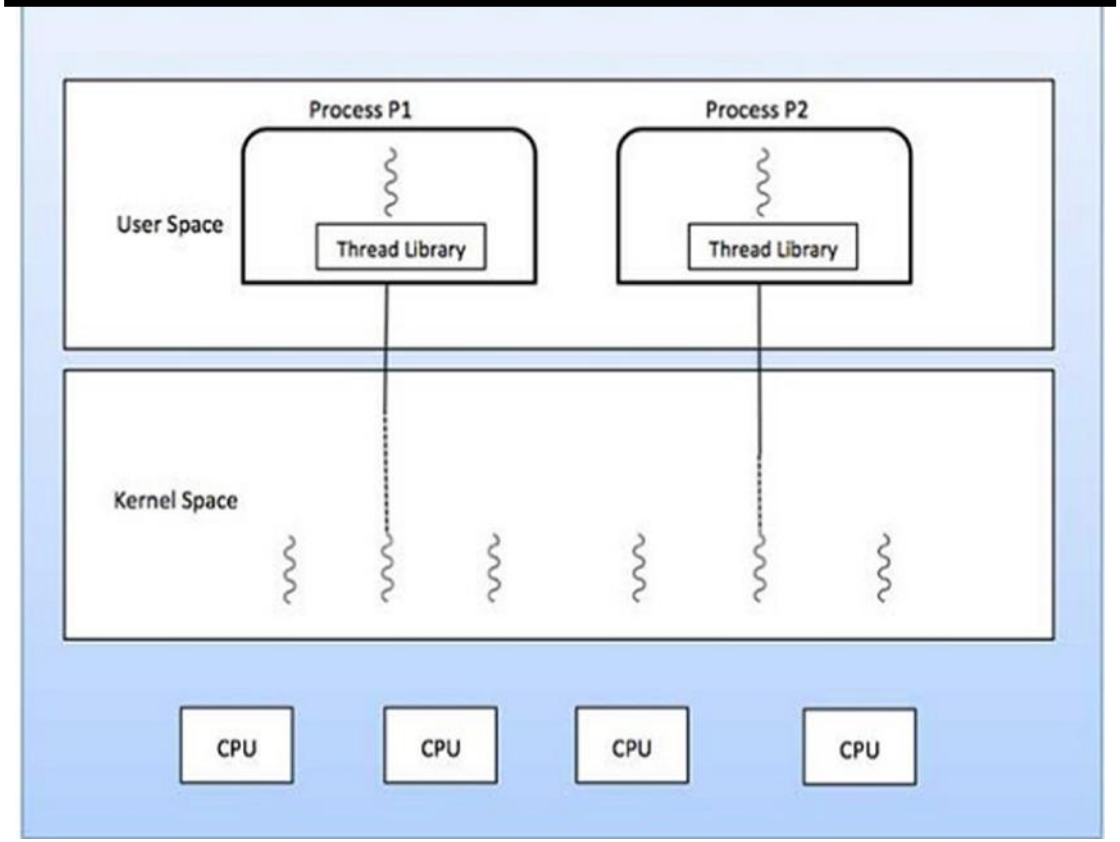
- Each user-level thread maps to kernel thread.
- It allows another thread to run when a thread makes a blocking system call.
- It allows multiple threads to run on parallel in multiprocessor system.

Drawback

- creating user threads requires creating the corresponding kernel threads.
- Examples

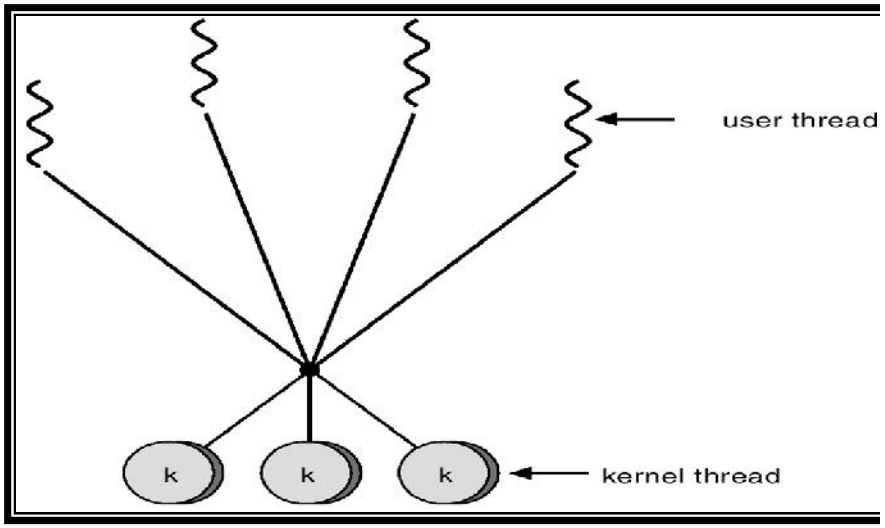
  - Windows , OS/2



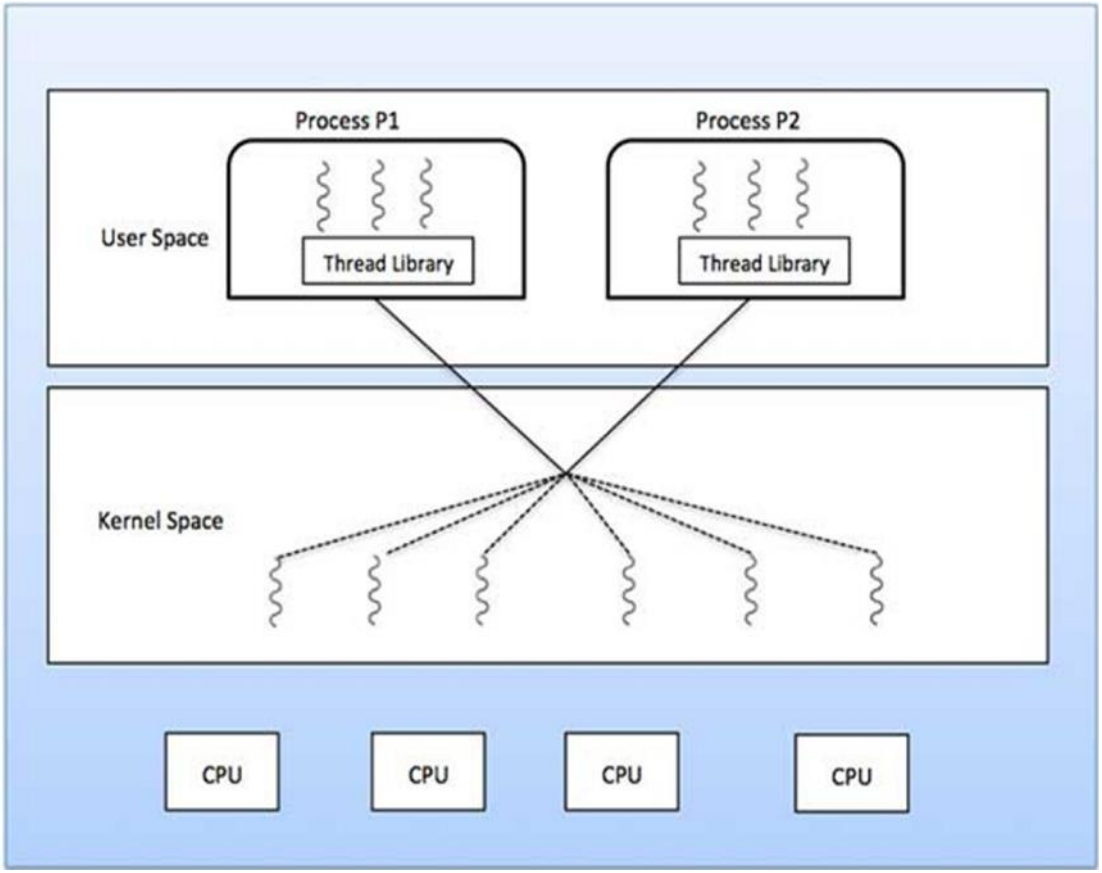


Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.



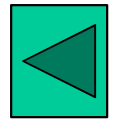
  - developers can create as many user threads as necessary and
  - the corresponding Kernel threads can run in parallel on a multiprocessor machine.
  - when a thread performs a blocking system call, the kernel can schedule another thread for execution.
  - Solaris 2, Windows...



Thread Usage

- A process has normally a single thread of control (execution sequence/flow).
  - Always at least one thread exists
  - If blocks, no activity can be done as part of the process
  - Better: be able to run concurrent activities (tasks) as part of the same process.
  - Now, a process can have multiple threads of control (multiple concurrent tasks).
- Threads run in pseudo-parallel manner (concurrently), share text and data

- Responsiveness
  - One thread blocks, another one runs.
  - One thread may always wait for the user
- Resource Sharing
  - Threads can easily share resources
- Economy
  - Creating a thread is fast
  - Context switching among threads may be faster
- Scalability
  - Multiprocessors can be utilized better



thread creation in C

```
//Example: thread.c
#include<stdio.h>
#include<pthread.h>
//thread function definition
void* threadFunction(void* args)
{
printf("I am a thread
function.\");
}
int main() {
  //creating thread id
pthread_t id;
  int ret;
//creating thread
ret=pthread_create(&id, NULL,
&threadFunction,NULL);
if(ret==0) {
printf("Thread created
successfully.\n);
}

else {
 printf("thread not
  created.\n);
    return 0;
   }
   printf("I am main
   function");
  return 0;
    }
gcc thread.c -o thread –
lpthread  ./thread
```

Java Threads

- As a Java programmer, you can choose between a single-threaded and a multithreaded programming paradigm.
- A single-threaded Java program has one entry point (the main() method) and one exit point.
  - All instructions are run serially, from start to finish.
- A multithreaded program has
  - a first entry point (the main() method), followed by
  - multiple entry and exit points for other methods that may be scheduled to run concurrently with the main() method.

Java Threads...

- Some of the reasons for using threads are that they can help to:
  - Make the UI more responsive
  - Take advantage of multiprocessor systems
  - Simplify modeling
  - Perform asynchronous or background processing

Java Threads...

- In Java, the support for threads is provided by two classes and one interface:
  - The java.lang.Thread class
  - The java.lang.Object class
  - The java.lang.Runnable interface
- Even a non-multithreaded program has one thread of execution, called the main thread.

Java Threads...

- In a multithreaded program, you can create other threads in addition to the main thread.
- You can write a thread class in one of two ways:
  - Extend the java.lang.Thread class
  - Implement the Runnable interface
- The Object class contains some methods that are used to manage the lifecycle of a thread.

Creating a Thread Using the Thread Class

//Example 1: How to define, instantiate, and start a thread by using the Thread class.

```
 public class ThreadTest {
public static void main(String[] args) {
   CounterThread ct = new CounterThread();
      ct.start();
 System.out.println("The thread has been
   started");}}
class CounterThread extends Thread {
       public void run() {
         for ( int i=1; i<=5; i++) {
System.out.println("Count: " + i); }} }
```

Creating a Thread Using the Thread Class

- Output

```
The thread has been started
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Creating a Thread Using the Thread Class

- Suppose you replace

  `ct.start()` by **`ct.run()`**

- In this case, no new thread would be started.
- The method **`run()`** will be executed in the main thread.

Output

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
The thread has been started
```

Creating a Thread Using the Runnable Interface

- You have seen in the previous section how to write a thread class by subclassing the **`java.lang.Thread`** class.

- But if your thread class already extends another class, it cannot extend the Thread class because Java supports only single inheritance.

- In this case, your thread class can implement the Runnable interface.

Creating a Thread Using the Runnable Interface

//Example 2: RunnableTest.java

```
class RunCounter extends Nothing implements
Runnable {
     public void run() {
          for ( int i=1; i<=5; i++) {
          System.out.println("Count: " + i);
               }}
      }
class Nothing {
          }
```

Creating a Thread Using the Runnable Interface

//Example 2: RunnableTest.java

```
public class RunnableTest {
 public static void main(String[] args) {
   RunCounter rct = new RunCounter();
   Thread th = new Thread(rct);
    th.start();
System.out.println("The thread has been
started");
         }
   }
```

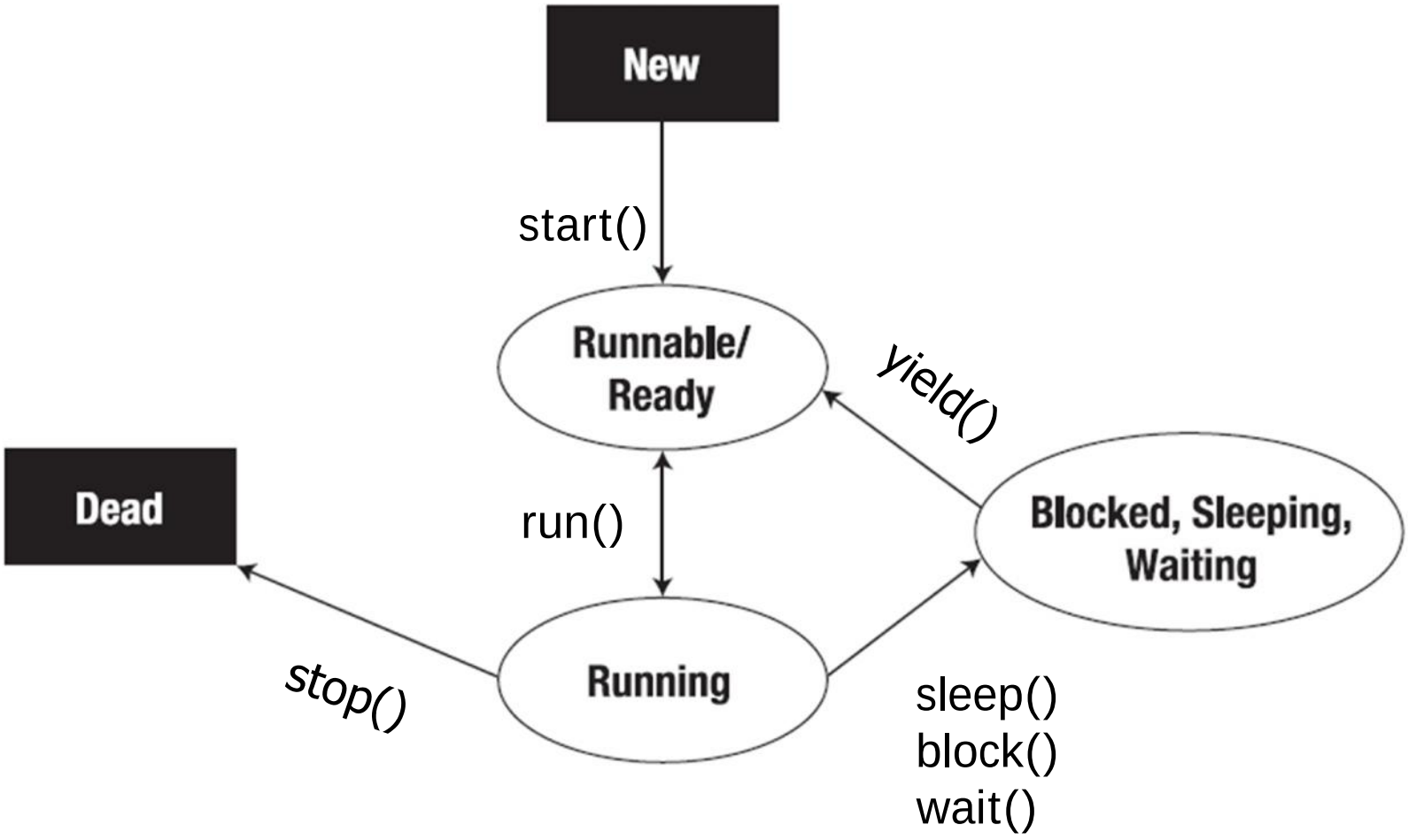
Creating a Thread Using the Runnable Interface

Output

```
The thread has been started
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

- You can accomplish the same task either:
  - by writing your thread class by extending the Thread class or
  - by implementing the Runnable interface.

Lifecycle of a Thread

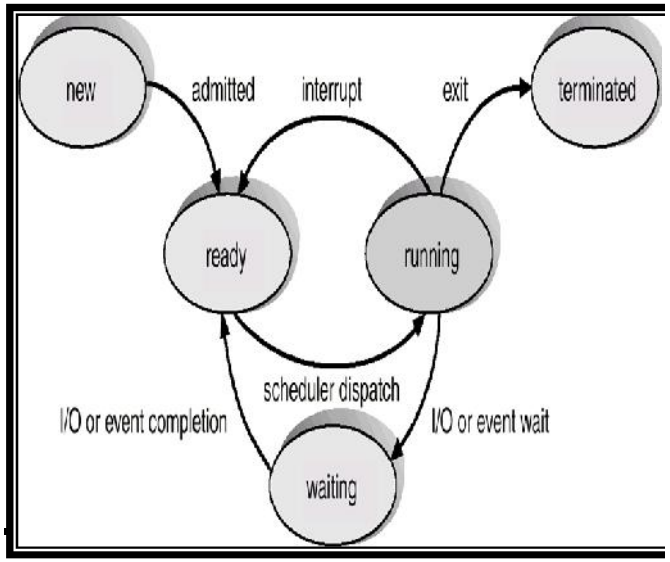


CPU Scheduling

- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms

CPU Scheduler: OS

- Selects among processes in memory that are ready to be executed, and allocates CPU to one of them.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state.
  2. Switches from running to ready state.
  3. Switches from waiting to ready.
  4. Terminates.



- Scheduling under 1 and 4 is nonpreemptive.
- All other scheduling is preemptive.

CPU Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – # of processes that complete their execution per unit time
- Turnaround time – total time required to execute a particular process
- Waiting time – amount of time a process waits in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced.

CPU Optimization Criteria

- Maximize throughput - run as many jobs as possible in a given amount of time.
  - This could be accomplished easily by running only short jobs or by running jobs without interruptions.
- Minimize response time - quickly turn around interactive requests.
  - This could be done by running only interactive jobs and letting the batch jobs wait until the interactive load ceases.
- Minimize turnaround time - move entire jobs in and out of the system quickly.
  - This could be done by running all batch jobs first (because batch jobs can be grouped to run more efficiently than interactive jobs).

CPU Optimization Criteria...

- Minimize waiting time - move jobs out of the READY queue as quickly as possible.
  - This could only be done by reducing the number of users allowed on the system so the CPU would be available immediately whenever a job entered the READY queue.
- Maximize CPU efficiency - keep the CPU busy 100 percent of the time.
  - This could be done by running only CPU-bound jobs (and not I/O-bound jobs).
- Ensure fairness for all jobs - give everyone an equal amount of CPU and I/O time.
  - This could be done by not giving special treatment to any job, regardless of its processing characteristics or priority.

Process Scheduling Algorithms

- Part of the operating system that makes scheduling decision is called scheduler and the algorithm it uses is called scheduling algorithm

- The Process Scheduler relies on a process scheduling algorithm,
  - based on a specific policy, to allocate the CPU and move jobs through the system.

- There are six process scheduling algorithms that have been used extensively.

First-Come, First-Served (FCFS) Scheduling

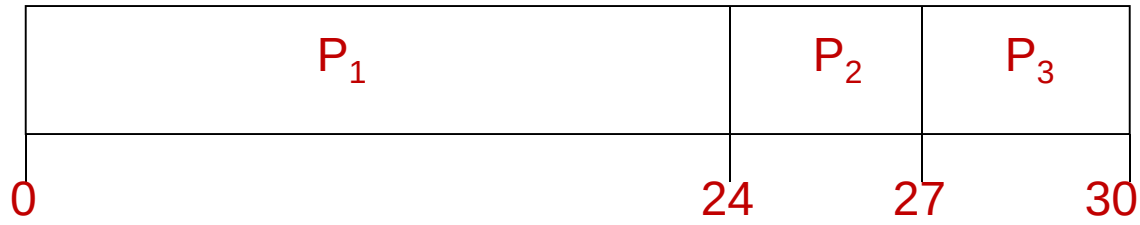
Basic Concept

- is a non preemptive scheduling algorithm that handles jobs according to their arrival time:
- the earlier they arrive, the sooner they're served.
- It's a very simple algorithm to implement because it uses a FIFO queue.
- In a strictly FCFS system there are no WAIT queues (each job is run to completion).

First-Come, First-Served (FCFS) Scheduling

Process	Burst Time (msec)
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17
- Turnaround time for P_1=24, P_2=27, P_3=30
- Avg turnaround time: (24+27+30)/3= 27

FCFS Scheduling (Cont.)

Suppose that the processes arrive in the order

$$P_2 , P_3 , P_1 .$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$
- Turnaround time for $P_1 = 30, P_2 = 3, P_3 = 6$
- Avg turnaround time: $(30+3+6)/3 = 13$
- Much better than previous case.

Implementation: FCFS

1. Input the processes along with their burst time (bt).
2. Find waiting time (wt) for all processes.
3. As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.
4. Find waiting time for all other processes i.e. for process i -> wt[i] = bt[i-1] + wt[i-1] .
5. Find turnaround time = waiting_time + burst_time for all processes.
6. Find average waiting time = total_waiting_time / no_of_processes.
7. Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

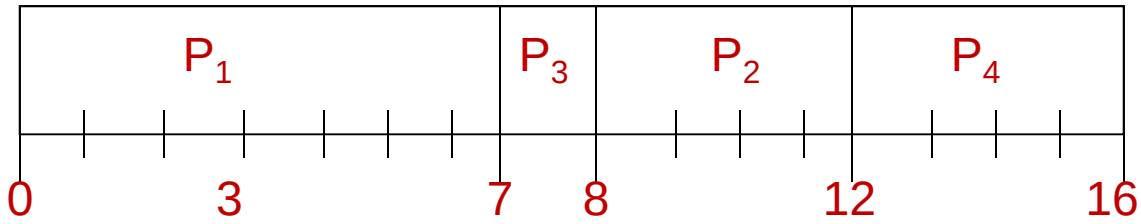
Shortest-Job-First (SJR) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time.
- Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).
- SJF is optimal – gives minimum average waiting time for a given set of processes.

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive)

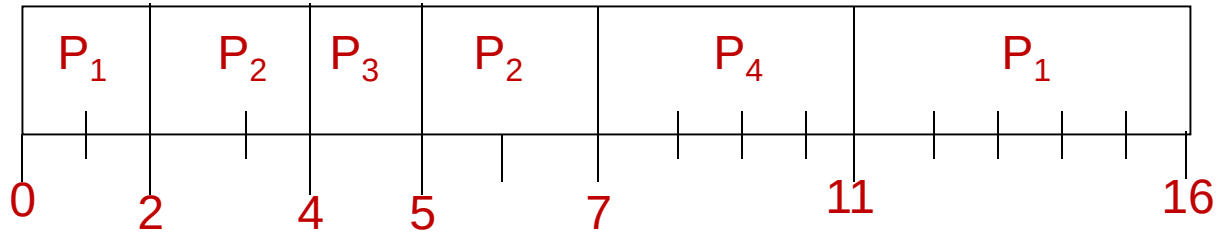


- Waiting time P_1=0-0=0, P_2=8-2=6, P_3=7-4=3, P_4=12-5=7
- Average waiting time = (0 + 6 + 3 + 7)/4 = 4
- Turnaround time: P_1= 7-0=7, P_2=12-2=10, P_3=8-4=4, P_4=16-5=11
- Avg. turnaround time: (7+10+4+11)/4= 8

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive)



- Average waiting time = (9 + 1 + 0 +2)/4 = 3
- Turnaround time:P_1=16-0=16, P_2=7-2=5, P_3=5-4=1, P_4=11-5=6
- Avg. turnaround time= (16+5+1+6)/4=7

Priority Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
  - Preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time.
- Problem ≡ Starvation – low priority processes may never execute.
- Solution ≡ Aging – as time progresses increase the priority of the process.

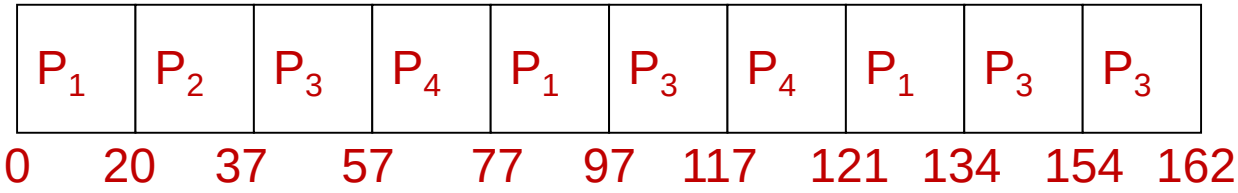
Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once.
- No process waits more than (n -1)q time units.
- Performance
  - q large $\Rightarrow$ FIFO
  - q small $\Rightarrow$ q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:



- Avg turnaround time = (134+37+162+121)/4 = 113.5
- Typically, higher average turnaround than SJF, but better response.

Project

1. program and simulate scheduling algorithms discussed before:
   - FCFS
   - Shortest-Job-First (SJR) Scheduling
   - Preemptive SJF
   - Priority Scheduling
   - Round Robin (RR)

- All scheduling algorithm shall be programmed in one file (all in one) so that once you submit no. of process, process's name and burst time, it should display options to see waiting time, turnaround time, avg. waiting and avg. turnaround time for scheduling alg. Presented above.